Полностью полиномиальная приближенная схема для задачи о "Рюкзаке"

Демидов Александр, ФИВТ ПМИ группа 296

Декабрь 2014

Содержание

1. Псевдополиномиальный алгоритм (динамическое программирование)
2. Построение полностью полиномиального приближенного алгоритма
3. Полный код алгоритма
4. Результаты тестовых запусков
5. Список литературы

1 Псевдополиномиальный алгоритм (динамическое программирование)

NP-полная задача "Knapsack" (о "Рюкзаке") формулируется следующим образом

Даны n предметов, для которых заданы соответственно веса $a_1, a_2, \ldots, a_n$ и стоимости $c_1, c_2, \ldots, c_n$, где $a_j, c_j \in \mathbb{N}\ \forall j \in \{1, 2, \ldots, n\}$. Размер рюкзака ограничен $B \in \mathbb{N}$.

Найти максимальное значение f^* функции

$$f = \sum_{i=1}^{n} c_i x_i,\ x_j \in \{0, 1\}$$

при условии

$$\sum_{i=1}^{n} a_i x_i \leq B$$

Рассмотрим простейшую схему решения задачи. На k-ом шаге будем обновлять 0/1-таблицу A размера $B \times f^*$, считая что для использования доступны только первые k предметов, и $\forall i, j\ A_{i,j} = 1$ соответствует достижимости состояния, а $A_{i,j} = 0$ недостижимости. После n-ого шага ненулевая ячейка с максимальной стоимостью соотвествует f^*.

Очевидный рекурсивный алгоритм дает сложность $O(nBf^*)$. Переборное решение можно улучшить, заметив, что из наборов с одинаковой стоимостью, но различной массой, достаточно учитывать только набор с меньшей массой. Таким образом, достаточно помнить не более f^* значений на каждой итерации. Время работы алгоритма составит в этом случае $O(nf^*)$. Подобный алгоритм будет полиномиален только для достаточно малых f^*, что зависит от исходных стоимостей предметов $c_1, c_2, \ldots, c_n$.

Ниже приводится реализация описанного псевдополиномиального алгоритма на языке *Python* 3.3

```python
# предметы (items) будем задавать
# следующим кортежем (item_cost, item_weight)
def knapsack_pseudopolinomial_solution(items, max_weigth):
    # ключ - цена набора
    # значение - [цена набора, вес набора]
    selection = {0: [0, 0]}

    for item in items:
        new_selection = []

        # по всем частичным решениям
        for solution in selection.values():
            new_solution = [solution[0] + item[0],  solution[1] + item[1]]

            # проверяем новое решение
            if (new_solution[1] <= max_weigth) \
                    and (new_solution[0] not in selection
                         or new_solution[1] < selection[new_solution[0]][1]):
                new_selection.append(new_solution)

        # регистрируем решения
        for solution in new_selection:
            selection[solution[0]] = solution

    return max(selection.keys())
```

2 Построение полностью полиномиального приближенного алгоритма

Определение 1. Будем называть **ϵ-оптимальным решением** допустимое решение со значением целевой функции, отличающимся от оптимального не более чем в $(1+\epsilon)$ раз.

Определение 2. Полностью полиномиальным приближенным алгоритмом будем называть алгоритм, находящий ϵ-оптимальное решение за время, полиномомиальное относительно длины входа и величины ϵ^{-1}.

Далее будет предложен вариант построения полностью полиномиального приближенного алгоритма для задачи "Knapsack" на основе полученного в предыдущем пункте псевдополиномиального алгоритма. Будем считать, что $\exists i : a_i \le B$, иначе ответ на задачу очевиден. Как было замечено, построенный алгоритм имеет сложность $O(nf^*)$, т. е. он будет полиномиален только для f^* ограниченных сверху некоторым полиномом. В случае больших значений f^* для уменьшения времени работы алгоритма можно попытаться некоторым образом изменить исходные стоимости предметов $c_1, c_2, \dots, c_n$.

Рассмотрим следующий вариант масштабирования:

$$(\hat{c}_i = \lfloor c_i/\alpha \rfloor * \alpha),\ \alpha \in \mathbb{Q}$$

Получена новая задача, оптимальный набор для которой не изменится, если разделить все стоимости $\hat{c}_i$ на величину α. В этом случае оценка времени работы алгоритма динамического программирования будет выглядеть следующим образом $O(nf^*/\alpha)$. При этом оптимум для новой задачи может отличаться от f^* в меньшую сторону, вследствие того что стоимости предметов могли стать меньше. Заметим также, что стоимость одного предмета не может уменьшиться более чем в α раз.

Далее нам потребуются следующие обозначения:
$\hat{x}_i$ — индикатор наличия предмета в оптимальном наборе новой задачи, $\hat{x}_i \in \{0, 1\}$
x_i^* — индикатор наличия предмета в оптимальном наборе исходной задачи, $x_i^* \in \{0, 1\}$
$\hat{f}$ — оптимум для новой задачи, $\hat{f} = \sum_{i=1}^n \hat{c}_i \hat{x}_i$

Выполнен ряд оценок:

$$\hat{f} = \sum_{i=1}^n \hat{c}_i \hat{x}_i \ge \sum_{i=1}^n \hat{c}_i x_i^* \ge \sum_{i=1}^n (c_i - \alpha) x_i^* \ge f^* - n\alpha$$

Нами получено отношение

$$f^* - \hat{f} \le n\alpha, \qquad (1)$$

где $f^* - \hat{f}$ можно рассматривать как показатель максимального отклонения $\hat{f}$ от оптимума исходной задачи f^*.

Имея в виду что $\hat{f} \le f^*$ получаем, что решение задачи после масштабирования стоимостей предметов является ϵ-оптимальным решением исходной задачи при выполнении следующего отношения

$$\hat{f} \ge f^*/(1 + \epsilon)$$

Отсюда и из (1) следует ограничение на α для существования ϵ-оптимального решения

$$\alpha \leq \frac{\epsilon f^*}{(1+\epsilon)n}$$

Полученное отношение усилится, если f^* заменить на его нижнюю оценку f^*_{low}. Одной из возможных оценок является тривиальная

$$f^*_{low} = c_{max} = \max_{i\,:\,a_i \leq B} c_i.$$

Тогда α определим как

$$\alpha = max\left\{1,\ \frac{\epsilon c_{max}}{(1+\epsilon)n}\right\}$$

При α, удовлетворяющем данному соотношению, решение задачи после масштабирования стоимостей предметов является ϵ-оптимальным решением исходной задачи. Посмотрим, как изменилась оценка времени работы при переходе к новым весам

$$O\left(\frac{n\hat{f}}{\alpha}\right) = O\left(\frac{nnc_{max}}{\alpha}\right) = O\left(\frac{nnc_{max}}{\frac{\epsilon c_{max}}{(1+\epsilon)n}}\right) = O\left(\frac{n^3(1+\epsilon)}{\epsilon}\right) = O\left(n^3/\epsilon\right)$$

Таким образом, получена схема построения полностью полиномиального приближенного алгоритма для задачи "Knapsack". Итоговый код алгоритма, а также результаты тестовых запусков программы приведены ниже.

3 Полный код алгоритма

Ниже приведена реализация на языке *Python* 3.3 полностью полиномиального приближенного алгоритма для задачи "Knapsack", построенного по описанной в предыдущем разделе схеме.

```
# псевдополиномиальное решение
# методом динамического программирования
def knapsack_pseudopolinomial_solution(items, max_weigth):
    # ключ - суммарная цена набора
    # значение - [цена набора, вес набора, номер последнего предмета]
    selection = {0: [0, 0, -1]}

    # сохраняем историю, для восстановления индексов наборов
    selection_history = [selection]

    for item in items:
        new_selection = []

        # по всем частичным решениям
        for solution in selection.values():
            new_solution = [solution[0] + item[0],  solution[1] + item[1], item[2]]

            # проверяем новое решение
            if (new_solution[1] <= max_weigth) \
                    and (new_solution[0] not in selection
                         or new_solution[1] < selection[new_solution[0]][1]):
                new_selection.append(new_solution)

        # регистрируем решения
        for solution in new_selection:
            selection[solution[0]] = solution

        # запоминаем текущие наборы для каждой стоимости
        selection_history.append(selection)

    # номера предметов оптимального набора
    indices = []

    solution = selection[max(selection.keys())]

    # восстановление набора
    while solution[2] != -1:
        indices.append(solution[2])
        solution = selection_history[solution[2]][solution[0] - items[solution[2]][0]]

    return indices


# полиномиальный приближенный алгоритм с точностью epsilon
def knapsack_approximated_solution(items, epsilon, max_weigth):
    # Вычисляем cmax
    low_cost = max(0, max(item[0] for item in items if item[1] <= max_weigth))

    if low_cost == 0:
        return 0

    # Вычисляем делитель для масштабирования весов
    alpha = max(1, epsilon * low_cost / (len(items) * (1 + epsilon)))

    # Масштабируем веса
    new_items = [(item[0]//alpha, item[1], item[2]) for item in items]

    # Вызываем решение динамическим программированием
    indices = knapsack_pseudopolinomial_solution(new_items, max_weigth)

    return sum([items[index][0] for index in indices])


if __name__ == "__main__":
```

```
    # список всех предметов
    items = []

    # точность
    epsilon = int(input())

    # ограничение размера рюкзака
    B = int(input())

    weights = input().split()
    costs = input().split()

    # в нижеприведенном коде предметы будем задавать
    # следующим кортежем (item_cost, item_weight, item_number)
    for i in range(len(costs)):
        items.append((int(costs[i]), int(weights[i]), i))

    print(knapsack_approximated_solution(items, epsilon, B))
```

4 Результаты тестовых запусков

Номер теста	Погрешность (ϵ)	Время работы полностью полиномиального приближенного алгоритма, ms	Время работы псевдополиномиального алгоритма, ms
1	0.15	1.1692358545815518	1.2775364539304136
2	0.15	28.30289881182437	39.37009086471613
3	0.15	51.52696145987155	60.606760523764285
3	0.2	40.50083124559548	60.606760523764285
4	0.15	96.12576419931212	105.77170336311857
4	0.2	72.67534816589599	105.77170336311857
5	0.2	70.92098111104096	90.83340647473337

Список литературы

1. Vazirani, Vijay. Approximation Algorithms. Springer-Verlag Berlin Heidelberg, 2003

2. Н.Н. Кузюрин, С.А. Фомин. Эффективные алгоритмы и сложность вычислений. МФТИ(ГУ), 2010

3. en.wikipedia.org the free encyclopedia